Casdoc: Unobtrusive Explanations in Code Examples

Mathieu Nassif
mnassif@cs.mcgill.ca
School of Computer Science
McGill University
Montréal, QC, Canada

Zara Horlacher
zara.horlacher@mail.mcgill.ca
School of Computer Science
McGill University
Montréal, QC, Canada

Martin P. Robillard
martin@cs.mcgill.ca
School of Computer Science
McGill University
Montréal, QC, Canada

ABSTRACT

Code examples are of great value to programmers trying to learn an unfamiliar API. Effective code examples are often surrounded with plain text explanations of the relevant concepts, techniques, and API elements involved in the example. However, authoring concise yet complete explanations is a challenging balancing act. To address this challenge, we propose Casdoc, a novel authoring technique and presentation format for annotated code examples. Casdoc-formatted code examples are HTML documents designed to embed unobtrusive explanations into the code. They thus contain more explanations to address the varying needs of a larger audience, without disrupting individual readers with information they already know. Explanations are split into short annotations and organized into an intuitive tree-like structure, thus supporting a streamlined authoring process. We used Casdoc to produce 105 Java code examples as part of the course material for an undergraduate computer science course. Students preferred the new format over traditional code examples. Their interaction with code examples suggests that the intuitive structure of Casdoc annotations reduces the need for navigation aids such as search fields.
On-line tool and video: https://www.cs.mcgill.ca/~martin/casdoc/

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools*; **Programming by example**; **Documentation**.

KEYWORDS

Software documentation, documentation format, code examples

ACM Reference Format:
Mathieu Nassif, Zara Horlacher, and Martin P. Robillard. 2022. Casdoc: Unobtrusive Explanations in Code Examples. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524610.3527730

1 INTRODUCTION

Documentation is the main resource programmers can use to discover and learn to use the application programming interface (API) of a framework or library. Indeed, low quality documentation can be a major obstacle for the adoption of a new framework [10]. One important aspect of API documentation is the inclusion of code examples [2, 8]. Prior work has proposed various techniques to generate them automatically (e.g., [3]). However, code examples alone are insufficient to convey all necessary knowledge to use an API effectively. Supporting knowledge such as their structure, key elements, and related concepts, is often provided in plain text explanations located around the code example [8].

Authoring explanations to complement code examples is a difficult balancing act. Too much content can bloat or fragment the documentation, but too little can fail to address the needs of some programmers. Anticipating different needs can lead to tangled explanations for unrelated tasks, making the relevant information for one task harder to find even when it is available [1, 15]. These issues are compounded by the reality that programmers with varying expertise will have different needs for the same documentation.

To address these seemingly conflicting issues, we propose a new format to improve the authoring and presentation of code examples. This new format, called Casdoc (for Cascading documentation), comprises three aspects. First, the explanations of a code example are broken down into short annotations. Second, these annotations are linked to the precise element of the code example or of another annotation they provide an explanation about. Finally, annotations are initially hidden and must be revealed by the reader.

This presentation format offers benefits to both readers and authors. Because readers reveal only information about the code elements and concepts they do not understand, they are not distracted by knowledge they are already familiar with. As a consequence, authors can include more content in a single document to address the needs of a larger audience, without worrying about the document becoming too cluttered. Furthermore, the need-oriented hierarchy of annotations provides an effective and consistent information structure for readers, while removing the burden of organizing documentation from authors.

Casdoc represents a departure from traditional formats of documentation that are designed around the limitations of printed media. It leverages the new possibilities of web technologies to structure information more intuitively, and only requires standard web technologies, such as any modern web browser, to present improved documents. Thus, Casdoc-formatted code examples can be embedded in any web-based documentation, such as tutorials, API reference documentation, and Q&A forums. We implemented a proof of concept tool to generate Casdoc documents for Java code examples. This tool takes as input Java files annotated using a domain-specific annotation language, and generates a self-contained Casdoc-formatted HTML document for each input file.

Readers can find examples of Casdoc documents, a short video introducing the main features of the documents, a user manual to annotate Java files, as well as a free on-line service to generate

ICPC '22, May 16–17, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9298-3/22/05…$15.00
https://doi.org/10.1145/3524610.3527730

```
String url = "jdbc:mysql://localhost/TUTORIALSPOINT";
String query = "SELECT * FROM Employees";
try(Connection conn = DriverManager.getConnection(url);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query);) {
  while (rs.next()) {
    System.out.println("ID: "+rs.getInt("id"));
    System.out.println("Name: "+rs.getString("name"));
  }
} catch (SQLException e) { e.printStackTrace(); }
```

Figure 1: Code example showcasing how to use the JDBC API to query an SQL database. Adapted from Tutorials Point [9].

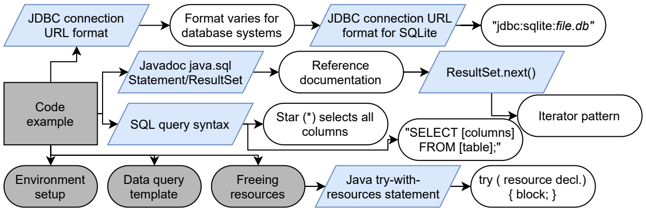

JDBC connection URL format
Format varies for database systems
JDBC connection URL format for SQLite
"jdbc:sqlite:*file.db*"
Javadoc java.sql Statement/ResultSet
Reference documentation
ResultSet.next()
Code example
SQL query syntax
Star (*) selects all columns
Iterator pattern
"SELECT [columns] FROM [table];"
Environment setup
Data query template
Freeing resources
Java try-with-resources statement
try (resource decl.) { block; }

Figure 2: Information needs (parallelograms) and their solution (white rounded rectangles) elicited by a code example of the JDBC API and its explanation (dark shapes). Casdoc can organize all of this information in a single document.

Casdoc documents from annotated Java files at https://www.cs.mcgill.ca/~martin/casdoc/

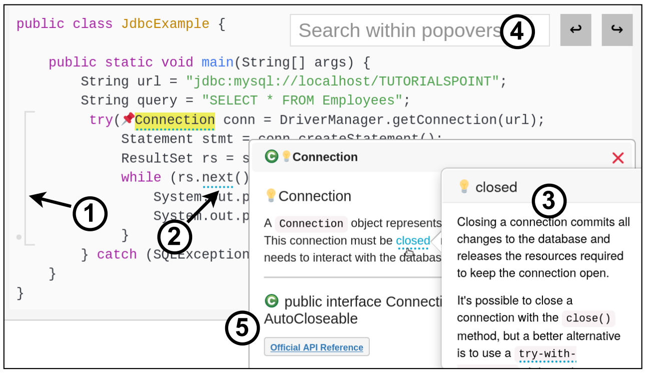
2 MOTIVATION

Let us consider a traditional search scenario for learning how to interact with an SQL database in Java. This scenario assumes prior experience in Java programming, but not with SQL databases.

We start with a web query such as *"Java SQL database example"*. Among the results, we find a code example similar to the one in Figure 1 that shows how to query a database. At this point, we realize our need to understand the connection URL and the syntax of SQL statements. Each information need triggers a new time-consuming search process, which can in turn trigger further searches. Figure 2 shows some of these additional searches that may be needed to fully understand the original code example.

This scenario shows that much effort can be spent searching for supporting information when trying to understand a code example. It also shows the *cascading* nature of information needs: Finding some information, for example that connection URLs follow vendor-specific formats, can create new information needs, i.e., the format for a given vendor. Needs also vary for different readers. Some may need to refresh their memory on, e.g., Java's try-with-resources statements, that other readers already know well [2].

All the supporting information could be contained in a single document. However, with a traditional format, doing so for a large audience of varying backgrounds can lead to large and intimidating documents, and programmers are more likely to miss the information they need when scanning many paragraphs. Documents with too much background information can also feel too verbose, inciting readers to look elsewhere for more concise documentation [1].



```
public class JdbcExample {

    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost/TUTORIALSPOINT";
        String query = "SELECT * FROM Employees";
         try(Connection conn = DriverManager.getConnection(url);
            Statement stmt = conn.createStatement();
            ResultSet rs = s
            while (rs.next()
                System.out.p
                System.out.p
            }
        } catch (SQLException
    }
}
```

Connection

Connection

A `Connection` object represents
This connection must be closed
needs to interact with the databas

public interface Connecti
AutoCloseable

closed

Closing a connection commits all changes to the database and releases the resources required to keep the connection open.

It's possible to close a connection with the `close()` method, but a better alternative is to use a `try-with-`

Figure 3: View of a Casdoc document, with block (1) and inline (2) anchors, revealed pop-up annotations (3), navigation aids (4), and automatically injected API documentation (5).

3 APPROACH

Our approach to present extensive descriptions of code examples effectively is to split explanations into concise hidden fragments, organized in a need-based tree structure, and to allow readers to reveal selected fragments. Thus, when opening a new document, only the uncluttered code example is shown, which is often what programmers first look for [2]. Readers can easily access just the additional information they need.

We developed this idea through a prototype implementation for Java code examples. The prototype consists of three components: a *documentation format* for HTML documents; a *markup language* to annotate code examples directly in Java files; and a *transformation tool* to convert annotated Java files to Casdoc code examples in HTML documents.

Because the focus of our work is to improve the *presentation* of code examples, we assume that the documentation *content* already exists or is written manually by an author. However, we designed Casdoc to allow extensions for generated documentation, so that it can benefit from state-of-the-art and future work on automated documentation generation.

3.1 Casdoc Documentation Format

We designed Casdoc based on the following design principles:

(1) Each document should express a clear, concise intent. Supplemental information should be unobtrusive.
(2) Navigation within the document should follow the information needs of readers. Only information relevant at a given time should be displayed.
(3) The format should support typical navigation actions such as *orienteering* and *teleporting* [12].
(4) Information added by the author of a document should be additive to existing tool-injected documentation, such as API reference documentation.

Figure 3 shows a partial view of a Casdoc document. Consistently with the first design principle, the initial view of the document consists only of the code example, with subtle *annotation markers*. An annotation marker indicates that the code element is an *anchor* to additional explanations. There are two kinds of anchors: a *block* anchor (1) matches an arbitrary block of consecutive lines, while an *in-line* anchor (2) matches a single keyword within a line of code. In Figure 3, a reader unfamiliar with ResultSet's next() method can see by the blue underline that this method is associated with further explanations.

Explanations are contained in *pop-up annotations* (number 3 in Figure 3). Hovering over an anchor reveals a *floating pop-up* that can be quickly opened and closed to clarify or recall existing knowledge [2]. If the reader wants to keep an annotation visible, they can *pin* it. Pinned pop-ups thus allow readers to lay out the document as they prefer. Annotations can themselves include in-line anchors for further explanations about, e.g., new concepts mentioned in the annotation. For example, in Figure 3, the right pop-up is a nested annotation that describes how to close a Connection object, an operation mentioned in the left pop-up. Hence, annotations are themselves concise, deferring their own supporting information to nested annotations. These choices arise from our second design principle. Readers can find further information by looking for markers near an unfamiliar element or concept, and display only the information they find useful.

Our third design principle led to the addition of navigation tools to support orienteering and teleporting actions [12] (number 4 in Figure 3). A search field allows readers to find deeply nested annotations and teleport to them. To help readers understand to which element an annotation relates to, nested annotations have breadcrumbs that show their parent annotation, i.e., the annotation that contains the anchor. Thus, readers using an orienteering strategy can navigate the annotations' tree structure backwards if they need to. Additionally, a pair of undo and redo buttons allows to navigate through the history of pinning and unpinning pop-up annotations.

Lastly, according to our fourth design principle, it should be possible to extend Casdoc to automatically insert some annotations in addition to the author's manual annotations. To avoid conflicts or undefined behavior, if both an author's and a tool-injected annotations have the same anchor, these annotations are combined into a single pop-up (number 5 in Figure 3). To avoid misleading information sources, an icon in the top left corner of pop-up boxes identifies the source of an annotation. We implemented one automated source of annotations: the API reference documentation of types and methods from the Java standard library. Because those automatically-generated annotations have predictable anchors, those anchors purposefully do not have visible markers.

3.2 Markup Language for Java Files

To generate Casdoc documents, authors need to insert annotations directly in Java files, which will be converted to HTML documents with our tool. Authors define annotations using a markup language that we designed to streamline the authoring process.[1]

Annotations are declared within Java block comments that begin with a question mark, i.e., enclosed in /*? ... */, which we refer to as *Casdoc comments*. This strategy—similar to documentation (Javadoc) comments enclosed in /** ... */—distinguishes the content of annotations from regular block comments to keep in the code example. A Casdoc comment can declare multiple annotations, and is placed immediately above the element it annotates.

```
/*?
 * Keyword: next
 * The next() method does two things: it checks if there are additional rows in the
 * ResultSet, and advances the cursor if there are.
 */
while (rs.next()) {
   System.out.println("ID: "+rs.getInt("id"));
   System.out.println("Name: "+rs.getString("name"));
}
```

Figure 4: Declaration of the annotation with an in-line anchor labelled "2" in Figure 3, placed within the Java source file.

Figure 4 shows an example of an annotation with the in-line anchor next. The first line of an annotation defines the type of anchor (e.g., the Keyword type indicates an in-line anchor) and the anchor itself. For annotations with a nested anchor, the second line declares the parent annotation's anchor. Annotations with a block anchor use the second line to declare the title, which appears at the top of the pop-up and is used for the breadcrumbs. Annotations with in-line or nested anchors use their anchor as title. The subsequent lines define the content of the annotation, using Markdown syntax. A detailed description of the Casdoc markup language is available on the tool's website.

The Casdoc markup language allows authors to create human-readable annotations in a Java file. Because annotations are inserted in block comments, authors can rely on any Java editor to create both the code examples and their annotations at the same time. Furthermore, because annotations are inserted directly above their anchors, authors can insert information exactly where it is relevant. Finally, in traditional documents with many paragraphs, authors must invest effort to ensure that the focus of each explanation is clear and that the narrative flow of the paragraphs is adequate. Casdoc mitigates this concern by clearly linking each explanation to its context, thus reducing the burden on authors.

3.3 Transformation Tool

The last component of our prototype is the tool to transform annotated Java files into HTML documents in Casdoc format. The generated HTML documents use only standard web technologies (CSS and JavaScript) and are entirely self-contained. Thus, once created, Casdoc documents can be shared and viewed without specialized viewing tools.

This transformation tool processes a Java file by parsing it into an abstract syntax tree (AST). It then extracts and removes the Casdoc comments, and places the remaining AST in a buffer to serve as the initial code example. The tool then parses the Casdoc comments to extract the annotations and the position of their respective anchors. It also generates annotations containing the API reference documentation for types and methods of the Java standard libraries.

After identifying the annotations, the tool updates the buffer to enclose each anchor in HTML tags, and appends the content of the annotations to the buffer. Finally, it injects the content of the

[1] In this article, we use the term *annotation* to refer exclusively to explanations of a code example, either in their comment form within Java files, or in their pop-up form in the HTML files. We do not use the term to refer to Java's annotation types.

buffer into a template HTML document with embedded CSS and JavaScript code that handles the display of pop-up annotations.

An interface to use the Casdoc transformation tool is available on the project website. The tool is free to use, but requires registration.

3.4 Limitations

We intended our prototype to support annotating any part of any Java file. However, for practical reasons, in-line anchors cannot span multiple lines, and manual annotations can only link to one anchor, even if the associated element is repeated in the code example. Furthermore, anchors within other Java comments proved challenging to properly design, so only block anchors are allowed within (non-Casdoc) comments. When annotating code examples (see Section 4), we found it was possible to work around these limitations without sacrificing the quality of the annotations.

4 EMPIRICAL ASSESSMENT

To gather insights into the benefits and limitations of Casdoc, we produced 105 annotated code examples for an undergraduate software design course in the Fall 2021 term, and we monitored how students interact with them.[2] These examples had been previously created by the third author as part of the development of the course, and we annotated them through the term. The examples showcased the use of good software design principles in Java. Most of them relied only on the Java standard libraries, and the others required either the JUnit testing library or JavaFX graphical user interface (GUI) library as a dependency. As a baseline, we also converted each Casdoc example into a static code example, in which annotations are shown as regular Java comments.

Creating the annotated code examples for a realistic context demonstrates the viability of our prototype. It was possible to inject in the code examples many clarifications for confusing elements, without diminishing the relative importance of the code example. Thus, because the authors did not have to consider which clarifications were worth documenting and how to structure these clarifications in a single narrative flow, it was possible to deliver more information in the code examples, without requiring more effort, as compared to authoring similar documents in a traditional format.

The interaction traces between participants and code examples also provided encouraging feedback about the quality of Casdoc documents. After removing traces from participants simply exploring Casdoc's features, we obtained data from 21 participants, who looked at 89 of the 105 code examples in total. Each of them consulted between 5 and 120 documents (between 4 and 59 unique documents), with an average of 22.7 (or 14.2 unique).

Originally, we had planned to compare the traces of participants using Casdoc to those preferring the traditional examples. However, only two participants tried the traditional format. One of them switched back to Casdoc within one minute. The other read some documents in the traditional format, but switched back after opening a code example with many annotations. Although this situation prevents a more thorough comparison of the two formats, it shows that most participants were favorable to Casdoc.

We also observed that participants most often hover over an anchor to reveal a floating pop-up rather than pin the pop-up. This observation suggests that readers may prefer information placed in elements that can quickly be revealed or hidden, possibly to avoid being distracted by this information once they understand it. Finally, we observed that participants did not rely much on the navigation tools we provided. In particular, only three of them used the search field, each only once. This could indicate that the structure of annotations is sufficiently intuitive that readers do not need to resort to typical navigation actions like teleporting and orienteering within the documents.

5 RELATED WORK AND CONCLUSION

Finding an optimal format for software documentation is not a new problem [5], yet many areas of the design space remain unexplored. The evolution of web technologies and the transition to on-line documentation created opportunities to design more effective formats [11]. For example, video tutorials are increasingly popular among developers [7]. Other technologies, such as augmented reality, can also provide new ways to improve documentation [4]. However, most of the recent work has focused on techniques to incorporate new information sources (e.g., [13]), to navigate existing documents (e.g., [14]), or to generate documentation (e.g., [6]). The question of how to *present* the generated or augmented documentation remains unsolved, and de facto standards remain close to the format of printed documents.

We presented Casdoc, a new interactive documentation format for code examples. Casdoc documents focus on the code example, hiding all other content from the initial view. They can contain many additional explanations, linked to the specific element of the code example they explain, that are revealed by readers if they need it. These explanations can be nested, e.g., to explain a new concept mentioned in another explanation, thus structuring documents as a tree of short information fragments rather than a sequence of paragraphs. This structure also streamlines the authoring process, encouraging authors to provide crisp explanations, without worrying about the organization of the document.

We implemented a prototype of Casdoc. With our tool, readers can annotate their own Java files to produce annotated code examples to include in their API learning material. When testing the viability of our prototype with undergraduate students, we found an overwhelming preference for our new format, preventing us from reliably comparing it to a standard static format. To address this gap, we designed a controlled experiment to compare how programmers react to Casdoc documents versus a baseline format. We plan to report the results of this experiment in future work.

ACKNOWLEDGMENTS

This work was funded by the Natural Sciences and Engineering Research Council of Canada and a Feng Qian undergraduate research award.

REFERENCES

[1] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Lineres-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*. 1199–1210.

[2] We only monitored the activity of students who provided informed consent. This study is approved by the Research Ethics Board of McGill University (file #21-06-007).

[2] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1589–1598.

[3] Raymond P. L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th IEEE/ACM International Conference on Software Engineering*. 357–367.

[4] Sridhar Chimalakonda and Akhila Sri Manasa Venigalla. 2020. Software Documentation and Augmented Reality: Love or Arranged Marriage?. In *Proceedings of the 28th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1529–1532.

[5] Bill Curtis, Sylvia B. Sheppard, Elizabeth Kruesi-Bailey, John Bailey, and Deborah A. Boehm-Davis. 1989. Experimental Evaluation of Software Documentation Formats. *Journal of Systems and Software* 9, 2 (1989), 167–207.

[6] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating Query-Specific class API Summaries. In *Proceedings of the 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 120–130.

[7] Laura MacLeod, Margaret-Anne Storey, and Andreas Bergen. 2015. Code, Camera, Action: How Software Developers Document and Share Program Knowledge Using YouTube. In *Proceedings of the IEEE 23rd International Conference on Program Comprehension*. 104–114.

[8] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What Makes a Good Code Example? A Study of Programming Q&A in StackOverflow. In *Proceedings of the 28th IEEE International Conference on Software Maintenance*. 25–34.

[9] Tutorials Point. 2009. JDBC - Sample, Example Code. https://www.tutorialspoint.com/jdbc/jdbc-sample-code.htm Last access: 2021-11-01.

[10] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (2011), 703–732.

[11] Philipp Schugerl, Juergen Rilling, and Philippe Charland. 2009. Beyond Generated Software Documentation – A Web 2.0 Perspective. In *Proceedings of the IEEE International Conference on Software Maintenance*. 547–550.

[12] Jaime Teevan, Christine Alvarado, Mark S. Ackerman, and David R. Karger. 2004. The Perfect Search Engine Is Not Enough: A Study of Orienteering Behavior in Directed Search. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 415–422.

[13] Christoph Treude and Martin P. Robillard. 2016. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering*. 392–403.

[14] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. 2015. Extracting Development Tasks to Navigate Software Documentation. *IEEE Transactions on Software Engineering* 41, 6 (2015), 565–581.

[15] Gias Uddin and Martin P. Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (2015), 68–75.